

Consistent Data Checkpoints in Distributed Database Systems: a Formal Approach*

Roberto BALDONI, Francesco QUAGLIA
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, Roma, Italy
baldoni,quaglia@dis.uniroma1.it

Michel RAYNAL
IRISA
Campus de Beaulieu
35042 Rennes-Cedex, France.
raynal@irisa.fr

Technical Report 22-97
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Luglio 1997

Abstract

Whether it is for audit or for recovery purposes, data checkpointing is an important problem of distributed database systems. Actually, transactions establish dependence relations on data checkpoints taken by data object managers. So, given an arbitrary set of data checkpoints (including at least a single data checkpoint from a data manager, and at most a data checkpoint from each data manager), an important question is the following one: "Can these data checkpoints be members of a same consistent global checkpoint?".

This paper answers this question by providing a necessary and sufficient condition suited for database systems. Moreover, to show the usefulness of this condition, two *non-intrusive* data checkpointing protocols are derived from this condition. It is also interesting to note that this paper, by exhibiting "correspondences", establishes a bridge between the data object/transaction model and the process/message-passing model.

1 Introduction

Checkpointing the state of a database is important for audit or recovery purposes. When compared to its counterpart in distributed systems, the database checkpointing problem has additionally to take into account the serialization order of the transactions that manipulates the data objects forming the database. Actually, transactions create dependencies among data objects which makes harder the problem of defining *consistent* global checkpoints in database systems.

Of course, it is always possible, in a database environment, to design a special transaction, that reads all data objects and saves their current values. The underlying concurrency control mechanism ensures this transaction gets a consistent state of the data objects. But this strategy is inefficient, intrusive (from the point of view of the concurrency control [13]) and not practical since, a read only transaction may take a very long time to execute and may cause intolerable delays for other transactions [10]. Moreover, as pointed out by Salem and Garcia-Molina [12], this strategy

*This work was done during a stay of Michel Raynal at the Dipartimento di Informatica e Sistemistica of Rome supported by a grant of the *Consiglio Nazionale delle Ricerche* in the context of the Short-Term-Mobility program.

may drastically increase the cost of rerunning aborted transactions. So, it is preferable to base global checkpointing (1) on local checkpoints of data objects taken by their managers, and (2) on a mechanism ensuring mutual consistency of local checkpoints (this will ensure that it will always be possible to get consistent global checkpoints by piecing together local checkpoints).

In this paper we are interested in exploiting such an approach. We consider a database in which each data object can be individually checkpointed (note that a data object could include, practically, a set of physical data items). If these checkpoints are taken in an independent way, there is a risk that no consistent global checkpoint can ever be formed (this leads to the well known *domino effect* [11]). So, some kind of coordination is necessary when local checkpoints are taken in order they be mutually consistent. In this paper we are interested in characterizing mutual consistency of local checkpoints. More precisely, we are interested in the two following points.

- First, we address the following question: “Given an arbitrary set S of checkpoints, can this set be extended to get a global checkpoint (i.e., a set including exactly one checkpoint from each data object) that is consistent?”. The answer to this question is well known when the set S includes exactly one checkpoint per data object [10]. It becomes far from being trivial, when the set S is incomplete, i.e., when it includes checkpoints from only a subset of data objects. When S includes a single data checkpoint, the previous question is equivalent to “Can this local checkpoint belong to a consistent global checkpoint?”.
- Then, we focus on data checkpointing protocols. Let us consider the property “Local checkpoint C belongs to a consistent global checkpoint”. We design two non-intrusive protocols. The first one ensures the previous property when C is any local checkpoint. The second one ensures it when C belongs to a predefined set of local checkpoints.

The paper consists of 4 main sections. Section 2 introduces the database model we consider in this paper. Section 3 defines consistency of global checkpoints. Section 4 answers the previous question. To provide such an answer, it studies the kind of dependencies both the transactions and their serialization order create among checkpoints of distinct data objects. More specifically, it is shown that, while some data checkpoint dependencies are causal, and consequently can be captured on the fly [8], some others are “hidden”, in the sense that, they cannot be revealed by causality. It is the existence of those hidden dependencies that actually makes non-trivial the answer to the previous question. Then, Section 5 shows how the necessary and sufficient condition stated in Section 4, can be used to design “transaction-induced” data checkpointing protocols ensuring the property “Local checkpoint C belongs to a consistent global checkpoint”. These protocols allow managers of data objects to take checkpoints independently on each other¹, and use transactions as a means to diffuse information, among data managers, encoding dependencies on the previous states of data objects. When a transaction that accessed a data object is committed, the data manager of this object may be directed to take a checkpoint in order previously taken checkpoints belong to consistent global checkpoints. Such a checkpoint is called *forced* checkpoint. This is done by the data manager which exploits both its local control data and the information exchanged at the transaction commit point.

Last but not least, this paper can be seen as a bridge between the area of distributed computing and the area of databases. For a long time, databases have provided distributed computing with very interesting problems and protocols related to data replication, concurrency control, etc. We show here how database checkpointing can benefit from studies that originated from distributed computing. Actually, a similar question has been addressed in the context of the asynchronous

¹These checkpoints are called *basic*. They can be taken, for example, during CPU idle time.

process/message-passing model [1, 9]. In this context a message establishes a simple relation between a pair of process local states. In the database context, a transaction may establish several relations between states of data objects. So, albeit there are some correspondences between the process/message-passing model and the data object/transaction model², it appears that extending process/message-passing model results to the context of database transactions is not trivial as a transaction is "something" more complicated than a message: a transaction is on several data objects at a time, and accesses them by read and write operations whose results depend on the serialization order.

2 Database Model

We consider a classical distributed database model. The system consists of a finite set of data objects, a set of transactions and a concurrency control mechanism (see [2, 6] for more details).

Data objects. Each data object is managed by a data manager DM . A set of data objects can be managed by the same data manager DM . For clarity, we suppose that the set of data managed by the same DM constitutes a single logical data. So, there is a data manager DM_x per data x .

Transactions. A transaction is defined as a partial order on *read* and *write* operations on data objects and terminates with a *commit* or an *abort* operation. $R_i(x)$ (resp. $W_i(x)$) denotes a read (resp. write) operation issued by transaction T_i on data object x . Each transaction is managed by an instance of the transaction manager (TM) that forwards its operations to the scheduler which runs a specific concurrency control protocol. The write set of a transaction is the set of all the data objects it wrote.

Concurrency control. A concurrency control protocol schedules read and write operations issued by transactions in such a way that any execution of transactions is *strict* and *serializable*. This is not a restriction as concurrency control mechanisms used in practice (e.g., two-phase locking 2PL and timestamp ordering) generate schedules ensuring both properties [3]. The *strictness* property states that no data object may be read or written until the transaction that currently writes it either commits or aborts. So, a transaction actually writes a data object at its commit point. Hence, at some abstract level, which is the one considered by our checkpointing mechanisms, transactions execute atomically at their commit points. If a transaction is aborted, strictness ensures no cascading aborts and the possibility to use *before images* for implementing abort operations which restore the value of an object before the transaction access. For example, a 2PL mechanism, that requires that transactions keep their write locks until they commit (or abort), generates such a behavior [3].

Distributed database. We consider a distributed database as a finite set of sites, each site containing one or several (logical) data objects. So, each site contains one or more data managers, and possibly an instance of the TM. TMs and DMs exchange messages on a communication network which is asynchronous (message transmission times are unpredictable but finite) and reliable (each message will eventually be delivered).

²At some abstraction level, there are similarities, on one side between processes and data objects, and on the other side, between messages and transactions (see Section 4.4).

Execution. Let $T = \{T_1, T_2, \dots, T_n\}$ be a set of transactions accessing a set $D = \{d_1, d_2, \dots, d_m\}$ of data objects (to simplify notations, data object d_i is identified by its index i). An execution E over T is a partial order on all read and write operations of the transactions belonging to T ; this partial order respects the order defined in each transaction. Moreover, let $<_x$ be the partial order defined on all operations accessing a data object x , i.e., $<_x$ orders all pairs of conflicting operations (two operations are conflicting if they access the same object and one of them is a write). Given an execution E defined over T , T is structured as a *partial order* $\hat{T} = (T, <_T)$ where $<_T$ is the following (classical) relation defined on T :

$$T_i <_T T_j \iff (i \neq j) \wedge (\exists x \Rightarrow (R_i(x) <_x W_j(x)) \vee (W_i(x) <_x W_j(x)) \vee (W_i(x) <_x R_j(x)))$$

3 Consistent Global Checkpoints

3.1 Local States and Their Relations

Each write on a data object x issued by a transaction defines a new version of x . Let σ_x^i denote the i -th version of x ; σ_x^i is called a *local state*. Transactions establish dependencies between local states. This can be formalized in the following way. When T_k issues a write operation $W_k(x)$, it moves the state of x from σ_x^i to σ_x^{i+1} . More precisely, σ_x^i and σ_x^{i+1} are the local states of x , just before and just after the execution³ of T_k , respectively. This can be expressed in the following way by extending the relation $<_T$ to include local states:

$$T_k \text{ changes } x \text{ from } \sigma_x^i \text{ to } \sigma_x^{i+1} \iff (\sigma_x^i <_T T_k) \wedge (T_k <_T \sigma_x^{i+1})$$

Let $<_T^+$ be the transitive closure of the extended relation $<_T$. When we consider only local states, we get the following *happened-before* relation denoted $<_{LS}$ (which is similar to Lamport's happened-relation defined on process events [8] in the process/message-passing model):

Definition 3.1 (*Precedence on local states, denoted $<_{LS}$*)

$$\sigma_x^i <_{LS} \sigma_y^j \iff \sigma_x^i <_T^+ \sigma_y^j$$

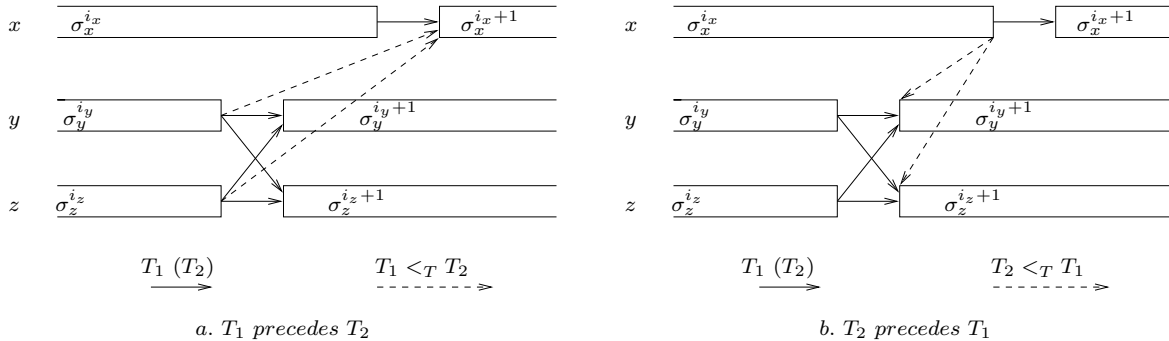


Figure 1: Partial Order on Local States

³Remind that, as we consider strict and serializable executions, “Just before and just after the execution of T_k ” means “Just before and just after T_k is committed”.

As the relation $<_T$ defined on transactions is a partial order, it is easy to see that the relation $<_{LS}$ defined on local states is also a partial order. Figure 1 shows examples of relation $<_{LS}$. It considers three data objects x , y , and z , and two transactions T_1 and T_2 . Transactions are defined in the following way:

$$\begin{aligned} T_1 &: R_1(x); W_1(y); W_1(z); \text{commit}_1 \\ T_2 &: R_2(y); W_2(x); \text{commit}_2 \end{aligned}$$

As there is a read-write conflict on x , two serialization orders are possible. Figure 1.a displays the case $T_1 <_T T_2$ while Figure 1.b displays the case $T_2 <_T T_1$. Each horizontal axis depicts the evolution of the state of a data object. For example, the second axis is devoted to the evolution of y : $\sigma_y^{i_y}$ and $\sigma_y^{i_y+1}$ are the states of y before and after T_1 , respectively.

Let us consider Figure 1.a. It shows that $W_1(y)$ and $W_1(z)$ add four pairs of local states to the relation $<_{LS}$, namely:

$$\begin{aligned} \sigma_y^{i_y} &<_{LS} \sigma_y^{i_y+1} \\ \sigma_z^{i_z} &<_{LS} \sigma_z^{i_z+1} \\ \sigma_x^{i_x} &<_{LS} \sigma_z^{i_z+1} \\ \sigma_z^{i_z} &<_{LS} \sigma_y^{i_y+1} \end{aligned}$$

Precedence on local states, due to write operations of transactions T_1 and T_2 , are indicated with continuous arrows, while the ones due to the serialization order are indicated in dashed arrows⁴. Figure 1.b shows which precedences are changed when the serialization order is reversed.

3.2 Consistent Global States

A *global state* of the database is a set of local states, one from each data object. A global state $G = \{\sigma_1^{i_1}, \sigma_2^{i_2}, \dots, \sigma_m^{i_m}\}$ is *consistent* if it does not contain two dependent local states, i.e., if:

$$\forall x, y \in [1, \dots, m] \Rightarrow \neg(\sigma_x^{i_x} <_{LS} \sigma_y^{i_y})$$

Let us consider again Figure 1.a. The three global states $(\sigma_x^{i_x}, \sigma_y^{i_y}, \sigma_z^{i_z})$, $(\sigma_x^{i_x}, \sigma_y^{i_y+1}, \sigma_z^{i_z+1})$ and $(\sigma_x^{i_x+1}, \sigma_y^{i_y+1}, \sigma_z^{i_z+1})$ are consistent. The global state $(\sigma_x^{i_x+1}, \sigma_y^{i_y}, \sigma_z^{i_z+1})$ is not consistent either because $\sigma_y^{i_y} <_{LS} \sigma_x^{i_x+1}$ (due to the fact $T_1 <_T T_2$) or because $\sigma_y^{i_y} <_{LS} \sigma_z^{i_z+1}$ (due to the fact T_1 writes both y and z). Intuitively, a non-consistent global state of the database is a global state that could not be seen by any omniscient observer of the database. It is possible to show that, as in the process/message-passing model, the set of all the consistent global states is a partial order [7].

3.3 Consistent Global Checkpoints

A *local checkpoint* (or equivalently a *data checkpoint*) of a data object x is a local state of x that as been saved in a safe place⁵ by the data manager of x . So, all the local checkpoints are local states, but only a subset of local states are defined as local checkpoints. Let C_x^i denote the i -th local checkpoint of x ; so, C_x^i corresponds to some σ_x^j with $i \leq j$. A *global checkpoint* is a set of local checkpoints one for each data object. It is *consistent* if it is a consistent global state.

We assume that all initial local states are checkpointed. Moreover, we also assume that, when we consider any point of an execution E , each data object will eventually be checkpointed.

⁴This shows dependencies between local states can be of two types. The ones that are due to each transaction taken individually, and the ones that are due to conflicting operations issued by distinct transactions (i.e., due to the serialization order).

⁵For example, if x is stored on a disk, a copy is saved on another disk.

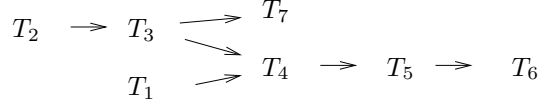


Figure 2: A Serialization Order

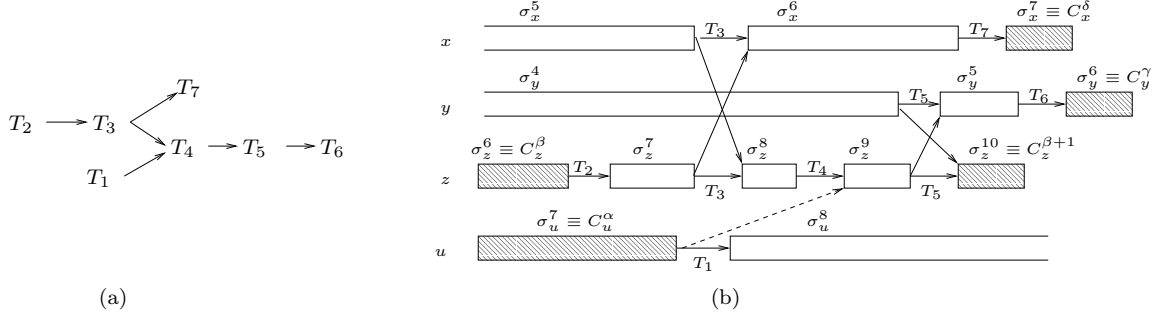


Figure 3: Data Checkpoint Dependencies

4 Dependence on Data Checkpoints

4.1 Introductory Example

As indicated in the previous section, due to write operations of each transaction, or due to the serialization order, transactions create dependencies among local states of data objects. Let us consider the following 7 transactions accessing data objects x , y , z and u :

$T_1 : R_1(u); W_1(u); \text{commit}_1$
 $T_2 : R_2(z); W_2(z); \text{commit}_2$
 $T_3 : R_3(z); W_3(z); W_3(x); \text{commit}_3$
 $T_4 : R_4(z); R_4(u); W_4(z); \text{commit}_4$
 $T_5 : R_5(z); W_5(y); W_5(z); \text{commit}_5$
 $T_6 : R_6(y); W_6(y); \text{commit}_6$
 $T_7 : R_7(x); W_7(x); \text{commit}_7$

Figure 2 depicts the serialization imposed by the concurrency control mechanism. Figure 3 describes dependencies between local states generated by this execution. Five local states are defined as data checkpoints (they are indicated by dark rectangles). We study dependencies between those data checkpoints. Let us first consider C_u^α and C_y^γ . C_u^α is the (checkpointed) state u before T_1 wrote it, while C_y^γ is the (checkpointed) state of y after T_6 wrote it (i.e., just after T_6 is committed). The serialization order (see Figure 2) shows that $T_1 <_T T_6$, and consequently $C_u^\alpha <_{LS} C_y^\gamma$, i.e., the data checkpoint C_y^γ is causally dependent [8] on the data checkpoint C_u^α (Figure 3 shows that there is a directed path of local states from C_u^α to C_y^γ). Now let us consider the pair of data checkpoints consisting of C_u^α and C_x^δ . Figure 3 shows that C_u^α precedes T_1 , and that C_x^δ follows T_7 . Figure 2 indicates that T_1 and T_7 are not connected in the serialization graph. So, there is no causal dependence between C_u^α and C_x^δ (Figure 3 shows that there is no directed path from C_u^α to C_x^δ).

But, as the reader can check, there is no consistent global checkpoint including both C_u^α and C_x^δ ⁽⁶⁾. So there is an *hidden* dependence between C_u^α and C_x^δ which prevents them to belong to the same consistent global checkpoint. We now provide a definition of dependence that takes into account both causal dependencies and hidden dependencies.

4.2 Dependence Path

Definition 4.1 (*Interval*) A checkpoint interval I_x^i is associated with data checkpoint C_x^i . It consists of all the local states σ_x^k such that:

$$(\sigma_x^k = C_x^i) \vee (C_x^i <_{LS} \sigma_x^k <_{LS} C_x^{i+1})$$

As an example, Figure 3 shows that I_z^β includes 4 consecutive local states of z . Note that, due to the assumptions on data checkpoints stated in Section 3.3, any local state belongs to exactly one interval. Let us call an edge of the partial order on local states a dependence edge.

Definition 4.2 (*Dependence Path*)⁷

There is a dependence path (DP) from a data checkpoint C_x^i to C_y^j (denoted $C_x^i \xrightarrow{DP} C_y^j$) iff:

- i) $x = y$ and $i < j$; or
- ii) there is a sequence (d_1, d_2, \dots, d_r) of dependence edges, such that:

- 1) d_1 starts after C_x^i ;
- 2) $\forall q: 1 \leq q < r, d_q$: let I_z^k be the interval in which d_q arrives; then d_{q+1} starts in the same or in a later interval (i.e., an interval I_z^h such that $k \leq h$)⁸;
- 3) d_r arrives before C_y^j .

4.3 Necessary and Sufficient Condition

Theorem 4.1 Let $\mathcal{I} \subseteq \{1, \dots, m\}$ and $\mathcal{S} = \{C_x^{i_x}\}_{x \in \mathcal{I}}$ be a set of data checkpoints. Then \mathcal{S} is a part of a consistent global checkpoint if and only if:

$$(\mathcal{P}) \quad \forall x, y \in \mathcal{I} : \neg(C_x^{i_x} \xrightarrow{DP} C_y^{i_y})$$

Proof

Sufficiency. We prove that if (\mathcal{P}) is satisfied then \mathcal{S} can be included in a consistent global checkpoint. Let us consider the global checkpoint defined as follows:

- if $x \in \mathcal{I}$, we take $C_x^{i_x}$;

⁶Adding C_y^γ and C_z^β to C_u^α and C_x^δ cannot produce a consistent global state as $C_z^\beta <_{LS} C_x^\delta$. Adding $C_z^{\beta+1}$ instead of C_z^β has the same effect as $C_u^\alpha <_{LS} C_z^{\beta+1}$.

⁷This definition generalizes the Z-path notion introduced in [9] for asynchronous message-passing systems. A Z-path is a sequence of messages. While a message is a “concrete entity”, a dependence edge is an “abstract entity”. See Section 4.4. So, as shown by the next theorem, the *dependence edge* abstraction allows to extend results of [9, 15, 1] to data checkpoints.

⁸Note that d_{q+1} can start before d_q arrives. This is where the dependence is “hidden”. If $\forall q$ d_{q+1} starts after d_q arrives, then, the dependence path (d_1, d_2, \dots, d_r) is purely causal.

- if $x \notin \mathcal{I}$, for each $y \in \mathcal{I}$ we consider the integer $m_x(y) = \min\{i \mid \neg(C_x^i \xrightarrow{DP} C_y^{i_y})\}$ (with $m_x(y) = 0$ if $i_y = 0$ or if this set is empty). Then we take $C_x^{i_x}$ with $i_x = \max_{y \in \mathcal{I}}(m_x(y))$. Let us note that, from that definition, it is possible that $i_x = 0$ (in that case, $C_x^{i_x}$ is an initial data checkpoint).

By construction, this global checkpoint satisfies the two following properties :

$$\forall x \notin \mathcal{I}, \forall y \in \mathcal{I} : \neg(C_x^{i_x} \xrightarrow{DP} C_y^{i_y}) \quad (1)$$

$$\forall x \notin \mathcal{I} \text{ such that } i_x > 0, \exists z \in \mathcal{I} : (i_z > 0) \wedge (C_x^{i_x-1} \xrightarrow{DP} C_z^{i_z}) \quad (2)$$

We show that $\{C_1^{i_1}, C_2^{i_2}, \dots, C_m^{i_m}\}$ is consistent. Assume the contrary. So, there exists x and y and a dependence edge d that starts after $C_x^{i_x}$ and arrives before $C_y^{i_y}$. So, it follows that:

$$(i_y > 0) \wedge (C_x^{i_x} \xrightarrow{DP} C_y^{i_y}) \quad (3)$$

Four cases have to be considered:

1. $x \in \mathcal{I}, y \in \mathcal{I}$. (3) is contradicted by assumption (\mathcal{P}).
2. $x \in \mathcal{I}, y \notin \mathcal{I}$. Since $i_y > 0$, from (2) we have: $\exists z \in \mathcal{I} : (i_z > 0) \wedge (C_y^{i_y-1} \xrightarrow{DP} C_z^{i_z})$.

As, at data x both the dependence edge ending the path $C_x^{i_x} \xrightarrow{DP} C_y^{i_y}$, and the dependence edge starting the path $C_y^{i_y-1} \xrightarrow{DP} C_z^{i_z}$ belong to the same interval, we conclude from (2) that $\exists z \in \mathcal{I} : (i_z > 0) \wedge (C_x^{i_x} \xrightarrow{DP} C_z^{i_z})$ which contradicts the assumption (\mathcal{P}).

3. $x \notin \mathcal{I}, y \in \mathcal{I}$. (3) contradicts (1).
4. $x \notin \mathcal{I}, y \notin \mathcal{I}$. Since $i_y > 0$, from (2) we have: $\exists z \in \mathcal{I} : (i_z > 0) \wedge (C_y^{i_y-1} \xrightarrow{DP} C_z^{i_z})$.

As in case 2, we can conclude that $\exists z \in \mathcal{I} : (i_z > 0) \wedge (C_x^{i_x} \xrightarrow{DP} C_z^{i_z})$ which contradicts (1).

Necessity. We prove that, if there is a consistent global checkpoint $\{C_1^{i_1}, C_2^{i_2}, \dots, C_n^{i_n}\}$ including \mathcal{S} , then property \mathcal{P} holds for any $\mathcal{I} \subseteq \{1, \dots, m\}$. Assume the contrary. So, there exist $x \in \mathcal{I}$ and $y \in \mathcal{I}$ such that $(C_x^{i_x} \xrightarrow{DP} C_y^{i_y})$. From the definition of \xrightarrow{DP} , there exists a sequence of dependence edges d_1, d_2, \dots, d_p such that:

$$\begin{aligned} & d_1 \text{ starts in } I_x^{i_x}, \\ & d_1 \text{ arrives after } I_{x_1}^{i_1}, \quad d_2 \text{ starts in } I_{x_1}^{j_1} \quad \text{with } i_1 \leq j_1 \\ & \dots \\ & d_{p-1} \text{ arrives in } I_{x_{p-1}}^{i_{p-1}}, \quad d_p \text{ starts in } I_{x_{p-1}}^{j_{p-1}} \quad \text{with } j_{p-1} \leq i_{p-1} \\ & d_p \text{ arrives in } I_y^{i_y-1} \end{aligned}$$

We show by induction on p that, $\forall t \geq i_y$, $C_x^{i_x}$ and C_y^t cannot belong to the same consistent global checkpoint.

Base step. $p = 1$. In this case, d_1 starts after $C_x^{i_x}$ and arrives before $C_y^{i_y}$, and consequently the pair $(C_x^{i_x}, C_y^{i_y})$ cannot belong to a consistent global checkpoint.

Induction step. We suppose the result true for some $p \geq 1$ and show that it holds for $p + 1$. We have:

$$\begin{aligned}
& d_1 \text{ starts in } I_x^{i_x}, \\
& \dots \\
& d_p \text{ arrives in } I_{x_p}^{i_p}, \quad d_{p+1} \text{ starts in } I_{x_p}^{j_p} \quad \text{with } i_p \leq j_p \\
& d_{p+1} \text{ arrives in } I_y^{i_y-1}
\end{aligned}$$

From the assumption induction applied to the path of dependence edges d_1, \dots, d_p , we have : for any $t \geq i_p + 1$, $C_x^{i_x}$ and $C_{x_p}^t$ cannot belong to the same consistent global checkpoint. Moreover, d_{p+1} starts in $I_{x_p}^{j_p}$ and arrives in $I_y^{i_y-1}$ imply that, for any $h \leq j_p$ and for any $t \geq i_y$, $C_{x_p}^h$ and C_y^t cannot belong to the same consistent checkpoint. Since $i_p \leq j_p$, it follows that no checkpoint of x_p can be included with $C_x^{i_x}$ and $C_y^{i_y}$ to form a consistent global checkpoint. \square

4.4 Database Systems vs Message-Passing Systems

Messages vs transactions. An analogous result for message-passing systems has been designed in [9] and generalized in [1]. As indicated in Section 4.2, point-to-point message-passing systems are characterized by the fact each message generates exactly one dependence edge between two process local checkpoints. In database systems, a dependence edge is due either to a write operation or to the serialization order. As a transaction may issue several write operations and is serialized in some order by the concurrency control mechanism, it follows that it may generate a lot of dependence edges between data checkpoints. For example, when a transaction writes α data objects, these writes establish α^2 dependence edges and supplementary edges are added according to the serialization order.

Consistency of a recovery line. Let us call *Recovery Line*⁹ a line joining all the data checkpoints of a global checkpoint. A recovery line is consistent iff the associated global checkpoint is consistent. Let us remind black and dashed arrows introduced in the example of Section 3.1: a black arrow denotes a local checkpoints precedence created by a transaction, while a dashed arrow denotes a local checkpoints precedence created by the serialization order. When considering such black and dashed arrows (see Figure 3), it is possible to show that a recovery line \mathcal{L} is consistent iff:

- No black arrow crosses \mathcal{L} .
- No dashed arrow crosses \mathcal{L} from the right of \mathcal{L} to the left of \mathcal{L} .

In a message-passing system, a recovery line (cut) is consistent iff no message crosses it from the right to the left [5]. Messages crossing the recovery line from left to right are “in-transit” with respect to the recovery line. This intuitively shows that, in a message-passing system: (1) a message corresponds to a “dashed arrow”, and (2) there is no “black arrow”. So, it appears that consistency of global checkpoints is a problem more involved in database systems than in message-passing systems.

5 Deriving “Transaction-Induced” Checkpointing Protocols

Required Properties. If we suppose that the set S includes only a checkpoint $C_x^{i_x}$, the previous Theorem leads to an interesting corollary \mathcal{C} :

⁹Also called *cut*, when adopting the distributed computing terminology.

Corollary 5.1 $C_x^{i_x}$ belongs to a consistent global checkpoint if and only if $\neg(C_x^{i_x} \xrightarrow{DP} C_x^{i_x})$.

Providing checkpointing protocols ensuring property \mathcal{C} is interesting for two reasons:

- (1) It avoids to waste time in taking a data checkpoint that will never be used in any consistent global checkpoint, and
- (2) No domino-effect can ever take place as any data checkpoint belongs to a consistent global checkpoint¹⁰.

Moreover, let us consider the following property \mathcal{P} : “If it exists, the set \mathcal{S}_n formed by the data checkpoints with the same index $n \geq 0$ (one from each data object), is a consistent global checkpoint”. In the following we provide two checkpointing protocols:

- The first protocol (\mathcal{A}) guarantees \mathcal{C} for all local checkpoints, and guarantees \mathcal{P} for any value of n .
- The second protocol (\mathcal{B}) ensures \mathcal{C} only for a subset of local checkpoints, and \mathcal{P} for some particular values of n .

Actually, those protocols can be seen as adaptations to the data-object/transactions model, of protocols developed for the process/message-passing model. More precisely, protocol \mathcal{A} corresponds with Briatico *et al.*’s protocol [4], while protocol \mathcal{B} corresponds with Wang-Fuchs’s protocol [14].

Local Control Variables. In both protocols we assume each data manager DM_x has an index i_x , which indicates the index (rank) of the last checkpoint of x (it is initialized to zero). Moreover, each data manager can take checkpoint independently (*basic checkpoints*), for example, by using a periodic algorithm which could be implemented by associating a timer with each data manager. A local timer is set whenever a checkpoint is taken. When a local timer expires, a basic checkpoint is taken by the data manager. Data managers are directed to take additional data checkpoints (*forced checkpoints*) in order to ensure \mathcal{C} or \mathcal{P} . The decision to take forced checkpoints is based on the control information piggybacked by transactions.

A protocol consists of two interacting parts. The first part, shared by both algorithms, specifies the checkpointing-related actions of transaction managers. The second part defines the rules data managers have to follow to take data checkpoints.

Protocols \mathcal{A} and \mathcal{B} : Behavior of a Transaction Manager. Let W_{T_i} be the write set of a transaction T_i managed by a transaction manager TM_i . We assume each time an operation of T_i is issued by TM_i to a data manager DM_x , it returns the value of x plus its index i_x . TM_i stores in M_{T_i} the maximum value among the indices of the data objects read or written by T_i . When transaction T_i is committed, the transaction manager TM_i sends a COMMIT message to each data manager DM_x involved in W_{T_i} . Such COMMIT messages piggyback M_{T_i} .

¹⁰When, after a crash, a data manager recovers, it can restore its last data checkpoint C . It follows from \mathcal{C} that C belongs to a consistent global checkpoint. So the database can be restarted as soon as each data manager has restored its data checkpoint contained in a consistent global checkpoint including C . Note that, when compared to message-passing systems, no “channel state” has to be restored.

Protocol \mathcal{A} : Behavior of a Data Manager. As far as checkpointing is concerned, the behavior of a data manager DM_x is defined by the two following procedures namely **take-basic-ckpt** and **take-forced-ckpt**. They defined the rules associated with checkpointing.

take-basic-ckpt(\mathcal{A}) :

When the timer expires:

- (AB1) $i_x \leftarrow i_x + 1$;
- (AB2) Take checkpoint $C_x^{i_x}$;
- (AB3) Reset the local timer.

take-forced-ckpt(\mathcal{A}) :

When DM_x **receives** COMMIT(M_{T_i}) **from** TM_i :

if $i_x < M_{T_i}$ **then**

- (A1) $i_x \leftarrow M_{T_i}$;
- (A2) Take a (forced) checkpoint $C_x^{i_x}$;
- (A3) Reset the local timer.

endif;

(A4) process the COMMIT message.

From the increase of the index i_x of a data object x , and from the rule **take-forced-ckpt(\mathcal{A})** (which forces a data checkpoint whenever $i_x < M_{T_i}$), the condition $\neg(C_x^{i_x} \xrightarrow{DP} C_x^{i_x})$ follows for any data checkpoint. Actually, this simple protocol ensures that, if $C_x^{i_x} \xrightarrow{DP} C_y^{i_y}$, then the index i_x associated with $C_x^{i_x}$ is strictly lesser than the index i_y associated with $C_y^{i_y}$.

It follows from the previous observation that if two data checkpoints have the same index, then they cannot be related by \xrightarrow{DP} . So, all the sets \mathcal{S}_n that exist are consistent. Note that the **take-forced-ckpt(\mathcal{A})** rule may produce gaps in the sequence of indices assigned to data checkpoints of a data object x . So, from a practical point of view, the following remark is interesting: when no data checkpoint of a data object x is indexed by a given value n , then the first data checkpoint of x whose index is greater than n , can be included in a set containing data checkpoints indexed by n , to form a consistent global checkpoint.

Protocol \mathcal{B} : Behavior of a Data Manager. This protocol introduces a system parameter $Z \geq 1$ known by all the data managers [14]. Only for subset of data checkpoints whose index is equal to $a \times Z$ (where $a \geq 0$ is an integer), we have: $\neg(C_x^{aZ} \xrightarrow{DP} C_x^{aZ})$. Moreover, when, $\forall x, C_x^{aZ}$ exists, then the global checkpoint \mathcal{S}_{aZ} exists and is consistent.

The rule **take-basic-ckpt(\mathcal{B})** is the same to the one of the protocol \mathcal{A} . In addition to the previous control variables, each data manager DM_x has an additional variable V_x , which is incremented by Z each time a data checkpoint indexed aZ is taken. The rule **take-forced-ckpt(\mathcal{B})** is the following.

take-forced-ckpt(\mathcal{B}) :

When DM_x **receives** COMMIT(M_{T_i}) **from** TM_i :

if $V_x < M_{T_i}$ **then**

- (B1) $i_x \leftarrow \lfloor M_{T_i}/Z \rfloor \times Z$;
- (B2) Take a (forced) checkpoint $C_x^{i_x}$;
- (B3) Reset the local timer;
- (B4) $V_x \leftarrow V_x + Z$.

endif;
 (B5) Process the COMMIT message.

About coordination. Compared to previous checkpointing protocols appeared in the literature [10, 13], which use an explicit coordination among data managers to get consistent global checkpoints, the proposed protocols provide the same result by using a *lazy* coordination which is propagated among data managers by transactions (with COMMIT messages). In particular, protocol \mathcal{A} starts a “transaction-induced” coordination each time a basic checkpoint is taken; while protocol \mathcal{B} starts a coordination each time a basic checkpoint, whose index is a multiple of the parameter Z , is taken. The latter protocol seems to be particularly interesting for database systems as it shows a tradeoff, mastered by a system parameter Z , between the number of forced checkpoints and the extent of rollback during a recovery phase. The greater Z is, the larger will be the rollback distance.

6 Conclusion

This paper has presented a formal approach for consistent data checkpoints in database systems. Given an arbitrary set of data checkpoints (including at least a single data checkpoint from a data manager, and at most a data checkpoint from each data manager), we answered the following important question “Can these data checkpoints be members of a same consistent global checkpoint?” by providing a necessary and sufficient condition. We have also derived two *non-intrusive* data checkpointing protocols from this condition; these checkpointing protocols use transactions as a means to diffuse information among data managers.

This paper can also be seen as a bridge between the area of distributed computing and the area of databases. We have shown that the checkpointing problem is harder in data-object/transaction systems than in process/message-passing systems. From a distributed computing point of view, we could say that database systems are difficult because they merge the “synchronous world” (every transaction taken individually has to be perceived as *atomic*: it can be seen as a multi-rendezvous among the objects it is on) and the “asynchronous world” (due to relations among transactions managed by the concurrency control mechanism).

References

- [1] Baldoni, R., H  l  ry, J.M. and Raynal, M., Consistent Records in Asynchronous Computations, *to appear in Acta Informatica*.
- [2] Bernstein, P.A., Hadzilacos, V. and Goodman, N., Concurrency Control and Recovery in Database systems, *Addison Wesley Publishing Co.*, Reading, MA, 1987.
- [3] Breitbart, Y., Georgakopoulos, D., Rusinkiewicz, M. and Silberschatz, A., On Rigorous Transaction Scheduling, *IEEE Transactions on Software Engineering*, 17(9):954-960, 1991.
- [4] Briatico, D., Ciuffoletti, A. and Simoncini, L., A Distributed Domino-Effect Free Recovery Algorithm, *in Proc. IEEE Int. Symposium on Reliability Distributed Software and Database*, pp. 207-215, 1984.
- [5] Chandy, K.M. and Lamport, L., Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.
- [6] Gray J.N. and Reuter A., Transaction Processing: Concepts and Techniques, *Morgan Kaufmann*, 1070 pages, 1993.

- [7] Hurfin, M., Plouzeau, N., and Raynal, M., On the Granularity of Events in Distributed Computations, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 13:115-124, 1994.
- [8] Lamport, L., Time, Clocks and The Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7):558-565, 1978.
- [9] Netzer, R.H.B. and Xu, J., Necessary and sufficient conditions for consistent global snapshots, *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165-169, 1995.
- [10] Pilarski, S. and Kameda, T., Checkpointing for Distributed Databases: Starting from the Basics, *IEEE Transactions on Parallel and Distributed Systems*, 3(5):602-610, 1992.
- [11] Randel, B., System structure for software fault tolerance, *IEEE Transactions on Software Engineering*, SE1(2):220-232, 1975.
- [12] Salem, K. and Garcia-Molina, H., Checkpointing Memory Resident Databases, *Tech. Rep. CS-TR-126-87*, Department of Computer Science, Princeton University, December 1987.
- [13] Son, S.H. and Agrawala, A.K., Distributed Checkpointing for Globally Consistent States of Databases, *IEEE Transactions on Software Engineering*, 15(10):1157-1166, 1989.
- [14] Wang, Y.M. and Fuchs, W.K., Lazy Checkpoint Coordination for Bounding Rollback Propagation, in *Proc. IEEE Int. Symposium on Reliable Distributed Systems*, pp. 78-85, 1993.
- [15] Wang, Y.M. Consistent Global Checkpoints That Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4):456-468, 1997.